02 BACKGROUND AND STATE OF THE ART

2.1 WebAssembly

The W3C publicly announced the WebAssembly language in 2017 as the four scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [**?**] in 2017. Wasm is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution [**?**].

Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop[1] written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version[2]. Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [**?** **?**]. Consequently, in 2019, the Bytecode Alliance [**?**] proposed WebAssembly System Interface (WASI) [**?**]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [**?**], making it suitable standalone and backend execution scenarios [**?** **?**].

2.1.2 WebAssembly's generation

WebAssembly programs are pre-compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For

[1] `https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw`

[2] `https://satoshinm.github.io/NetCraft/`

instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code.

In Listing 2.1 and Listing 2.2, we illustrate a C program and its corresponding Wasm binary. The C function includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses. The Wasm code in Listing 2.2 displays the textual format of the generated Wasm (Wat).

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.1: Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
; Type section: 01 ... ;
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
; Import section: 02 ... ;
  (import "env" "ftoi" (func $ftoi (type 0)))
; Code section: 03 ... ;
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block  ;label = @1;
      loop  ;label = @2;
        i32.const 0
        local.get 0
        i32.add
        i32.load
        local.tee 1
        i32.const 101
        i32.ge_s
        br_if 1 ;@1;
        local.get 0
        i32.const 4
        i32.add
        local.tee 0
        br_if 0 ;@2;
      end
      i32.const 0
      return
    end
    f32.const 0x1.9147aep+3
    call $ftoi
    local.get 1
    i32.add)
; Memory section: 05 ... ;
  (memory (;0;) 1)
; Global section: 06 ... ;
  (global (;4;) i32 (i32.const 1000))
; Export section: 07 ... ;
  (export "memory" (memory 0))
  (export "A" (global 2))
; Data section: 0d ... ;
  (data $data (0) "\00\00\00\00...")
)
```

Listing 2.2: Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.

2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized. Thus, its consuming process typically involves a straightforward one-to-one mapping. For example, a compiler might accelerate the compilation process by parallelizing the parsing process. The main reason behind this claim is that a Wasm binary is organized into a contiguous collection of sections. In Figure 2.1 we show the

binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by an 8-byte section size, and concludes with the section content, which precisely matches the size indicated earlier.

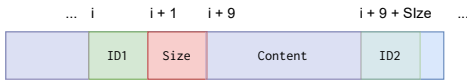

Figure 2.1: Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.

A Wasm binary contains sections of 12 types, each with a specific semantic role and placement within the module. Each section is optional, where an omitted section is considered empty. In the following text, we summarize each one of the 12 types of Wasm sections, providing their name, ID, and purpose. In addition, some sections are annotated as comments in the Wasm code in Listing 2.2.
TODO Check if better with examples in each section

Custom Section (`00`) : Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary. This type of section has no order constraints with other sections and is optional. Compilers usually skip this section when consuming a WebAssembly binary.

Type Section (`01`) : Contains the function signatures for functions declared or defined within the binary. It must occur only once in a binary. It can be empty.

Import Section (`02`) : Lists elements imported from the host, including functions, memories, globals, and tables. It must occur only once in a binary. It can be empty.

Function Section (`03`) : Details functions defined within the binary. It essentially maps Type section entries to Code section entries. It must occur only once in a binary. It can be empty.

Table Section (`04`) : Groups functions with identical signatures to control indirect calls. It must occur only once in a binary. It can be empty.

Memory Section (`05`) : Specifies the number and initial size of unmanaged linear memories. It must occur only once in a binary. It can be empty.

Global Section (`06`) : Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary. It must occur only once in a binary. It can be empty.

Export Section (`07`) : Declares elements like functions, globals, memories, and tables for host engine access. The entry point of the WebAssembly binary is typically declared here. It must occur only once in a binary. It can be empty.

Start Section (`08`) : Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions. It must occur only once in a binary. It can be empty.

Element Section (`09`) : Contains elements to initialize the binary tables. It must occur only once in a binary. It can be empty.

Code Section (`10`) : Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions. It must occur only once in a binary. It can be empty.

Data Section (`11`) : Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memory. It must occur only once in a binary. It can be empty.

Data Count Section (`12`) : Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed. It must occur only once in a binary. It can be empty.

■ 2.1.4 WebAssembly's runtime structure

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Module Instances, Table Instances, Export Instances, Import Instances, the Execution Stack, Memory Instances, Global Instances, Function Instances and Locals. These components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a brief description of each runtime component. Notice that, the runtime structure is an abstraction that serves to validate the execution of a Wasm binary.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module. It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module's exports and imports.

Table instances : A table instance is a vector of function elements. WebAssembly tables are used to support indirect function calls. For example, it allows modeling dynamic calls of functions (through pointers) from languages

such as C/C++, for which the Wasm's compiler is in charge of populating the static table of functions.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a Wasm binary to the host environment.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host environment.

The Execution Stack holds typed values and control frames, with control frames handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function invocation. As globals, locals are part of the managed data.

Note 1. *Along with this dissertation, as the work of Lehmann et al. [**?**], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.*

Function Instances : A function instance groups locals and a function body. Locals are typed variables that are local to a specific function invocation. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back

the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at result. As the listing shows, instructions are annotated with a numeric type.

2.1.5 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.

```
block                                    loop
   block                                    ...
      br 1    ; Jump instructions          br 0    ;first-order break;
              are annotated with the        ...
              depth of the block they    end    ; end instructions break
               jump to;                         the block and jump to next
   end                                          instruction;
end                                      ...
...
```

Listing 2.3: Example of breaking a block and a loop in WebAssembly.

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. For the purposes of this dissertation, we introduce a specific term to describe a particular kind of break within loops:

Definition 1. *Break instructions within loops that effectively jump to the loop's beginning are termed* first-order breaks.

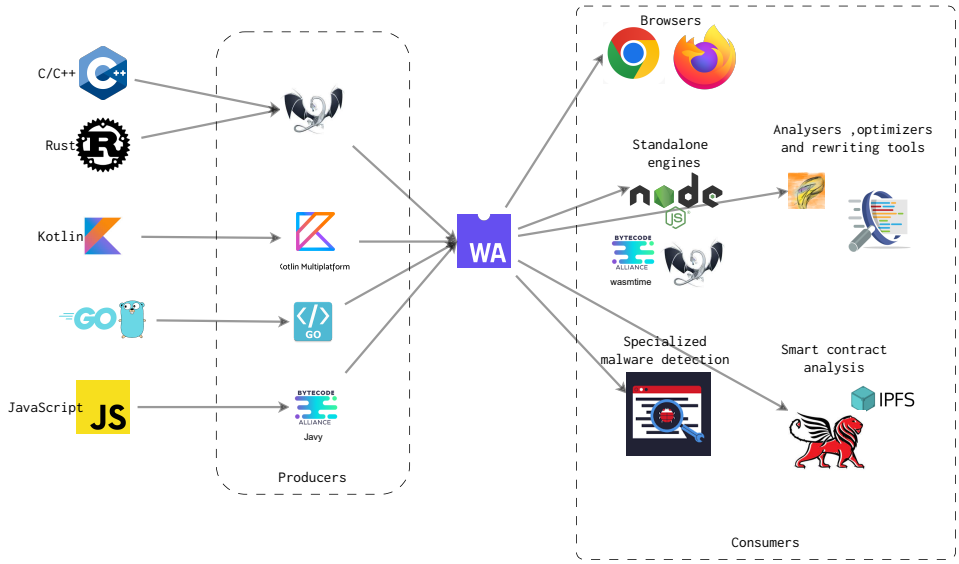

Figure 2.2: WebAssembly's ecosystem landscape separated into producers and consumers. For the sake of simplicity we do not include each tool mentioned in this chapter.

2.1.6 WebAssembly's ecosystem

WebAssembly programs are designed for execution in host environments such as web browsers. Though the execution of a WebAssembly program might be considered its final lifecycle stage, the WebAssembly ecosystem is far from simplistic. It comprises multiple stakeholders and a rich array of tools that cater to various needs [**?**]. In Figure 2.2 we simplify the ecosystem landscape by separating it into producers, consumers and major stakeholders categories. In the subsequent text, we describe the WebAssembly ecosystem by separating it into stakeholders categories.

Producers , such as compilers, transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release[3], supporting a diverse set of frontend languages like C/C++, Rust, Go, and AssemblyScript[4]. In parallel developments, the KMM framework[5] has incorporated WebAssembly as a compilation target, and the Javy approach[6] focuses on encapsulating JavaScript code within isolated WebAssembly binaries.

[3] `https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0`
[4] A subset of the TypeScript language
[5] `https://kotlinlang.org/docs/wasm-overview.html`
[6] `https://github.com/bytecodealliance/javy`

This is achieved by porting both the engine and the source code into a secure WebAssembly environment. Blazor also enables the compilation of C code into WebAssembly binaries for browser execution[7]. Regardless of the source language or framework, the resulting WebAssembly binary functions similar to a traditional shared library, replete with code instructions, symbols, and exported functions.

Consumers encompass tools that undertake the tasks of validating, analyzing, optimizing, transpiling to machine code, and executing WebAssembly binaries, e.g. browser clients. In the text that follows, we dissect them into specific categories and their respective domains of application. Notice that, while some tools are designed for a specific domain, others are more general-purpose and might encapsulate more than one task in the WebAssembly ecosystem. For example, this is the case of browsers and standalone engines, which in one way or the other perform each one of the previous tasks.

Browser engines like V8[8] and SpiderMonkey[9] are at the forefront of executing WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.3. For example, V8 just employs quick, rudimentary optimizations, such as constant folding and dead code removal, to guarantee fast readiness for a Wasm binary to execute.[10]

Standalone engines: WebAssembly has expanded beyond browser environments, largely due to the WASI[**?**]. It standardizes the interactions between host environments and WebAssembly modules through a POSIX-like interface. A range of standalone engines like WASM3[11], Wasmer[12], Wasmtime[13], WAVM[14], and Sledge[**?**] have emerged to support WebAssembly and WASI. In a similar vein, Singh et al.[**?**] introduced a virtual machine for WebAssembly tailored for Arduino-based devices. Salim et al.[**?**] proposed TruffleWasm, an implementation of WebAssembly hosted on Truffle and GraalVM. Additionally, SWAM[15] stands out as WebAssembly interpreter implemented in Scala. Finally, WaVe[**?**] offers a WebAssembly interpreter featuring mechanized verification of the WebAssembly-WASI interaction with the underlying operating system.

[7] `https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor`
[8] `https://chromium.googlesource.com/v8/v8.git`
[9] `https://spidermonkey.dev/`
[10] This analysis was corroborated through discussions with the V8 development team and through empirical studies in one of our contributions[**?**]
[11] `https://github.com/wasm3/wasm3`
[12] `https://wasmer.io/`
[13] `https://github.com/bytecodealliance/wasmtime`
[14] `https://github.com/WAVM/WAVM`
[15] `https://github.com/satabin/swam`

Static and dynamic analysis, optimization and validation: As the WebAssembly ecosystem continues to grow, the need for robust tools to ensure its security and reliability has increased. To address this, a variety of tools have been developed that employ different strategies to identify vulnerabilities in WebAssembly programs. Tools like Wassail[**?**], SecWasm[**?**], Wasmati[**?**], WasmA[**?**], and Wasp[**?**] leverage techniques such as information flow control, code property graphs, control flow analysis, and concolic execution. VeriWasm[**?**] stands out as a static offline verifier specifically designed for native x86-64 binaries compiled from WebAssembly. In the realm of dynamic analysis, tools like TaintAssembly[**?**], Wasabi[**?**], and Fuzzm[**?**] offer similar functionalities. Hybrid methods have also gained traction, with tools like CT-Wasm[**?**] enabling the verifiably secure implementation of cryptographic algorithms in WebAssembly. Binaryen[16] serves as a comprehensive toolkit for WebAssembly binary manipulation, including validation, optimization, and compilation to machine code. Stiévenart and colleagues have introduced a dynamic approach to slice WebAssembly programs based on Observational-Based Slicing (ORBS)[**?** **?**]. Finally, Wafl[**?**] extends AFL++ to perform coverage-based fuzzing on WebAssembly binaries.

Specialized Malware Detection In niche areas like cryptomalware detection, tools like MineSweeper[**?**], MinerRay[**?**], and MINOS[**?**] utilize static analysis techniques. Conversely, dynamic analysis is the forte of tools like SEISMIC[**?**], RAPID[**?**], and OUTGuard[**?**].

Smart Contract Analysis In the field of smart contracts, static analysis tools like WANA[**?**], and EOSAFE[**?**] are employed to unearth vulnerabilities in WebAssembly smart contracts. Dynamic analysis tools in this sphere include EOSFuzzer[**?**] and wasai[**?**]. Similarly, Manticore[17] supports the symbolic execution of Wasm smart contracts.

Binary rewriting tools and obfuscators The landscape for tools that can modify, obfuscate, or enhance WebAssembly binaries for various purposes—including has increased. For instance, BREWasm[**?**] provides a comprehensive static binary rewriting framework specifically designed for WebAssembly. Wobfuscator[**?**] takes a different approach, serving as an opportunistic obfuscator for Wasm-JS programs. Madvex[**?**] focuses on modifying WebAssembly binaries to evade malware detection, with its approach being limited to alterations in the code section of a WebAssembly binary. Additionally, WASMixer[**?**] obfuscates WebAssembly binaries, by including memory access encryption, control flow flattening, and the insertion of opaque predicates.

[16] `https://github.com/WebAssembly/binaryen`
[17] `https://github.com/trailofbits/manticore/tree/master/manticore`

2.1.7 WebAssembly opportunities

Despite its robust design, the WebAssembly's ecosystem is still nascent and faces several security vulnerabilities, both in its implementation and in the binaries it produces. As outlined, WebAssembly is deterministic, well-typed, and follows a structured control flow, making it easier for compilers and engines to sandbox its execution[**?**]. However, vulnerabilities persist. For instance, Genkin et al. demonstrated that WebAssembly could be exploited to exfiltrate data using cache timing-side channels[**?**]. Lehmann et al. and Stiévenart and colleagues show that vulnerabilities in C/C++ source code could propagate to WebAssembly binaries[**? ?**].

Moreover, the WebAssembly ecosystem is still in its infancy compared to more mature programming environments. A 2021 study by Hilbig et al. found only 8,000 unique WebAssembly binaries globally[**?**], a fraction of the 1.5 million and 1.7 million packages available in npm and PyPI, respectively. This limited dataset poses challenges for machine learning-based analysis tools, which require extensive data for effective training. The scarcity of WebAssembly programs also exacerbates the problem of software monoculture, increasing the risk of compromised WebAssembly programs being consumed[**?**]. This dissertation aims to mitigate these issues by introducing a comprehensive suite of tools designed to enhance WebAssembly security through Software Diversification and to improve testing rigor within the ecosystem.

2.2 Software diversification

2.2.2 Generating Software Diversification

2.2.3 Variants generation

2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. `https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13`

2.3 Exploiting Software Diversification

2.3.2 Defensive Diversification

2.3.3 Offensive Diversification